Session Code: TLS400

Whidbey

tools & languages

Visual C++ “Whidbey”: Advanced Code Generation

Kang Su Gatlin
VC++ Program Manager
Microsoft Corporation
kanggatl@microsoft.com

PDC03

Make the connection

Top 5 Reasons Why You're At This Talk

- Your inner-loop took six cycles per iteration instead of four… and that gets you mad
- You're the **"wolf"**… the one they bring in when performance counts
- You realize saving 10 seconds in a program used by a million users is 115 days saved
- You read Itanium assembly for fun… obviously this talk looked interesting
- You care about generating good code… and that is what Visual C++ does

Agenda

- A Very Quick Refresher Course
- WPO & PGO
- Multithreading Support
- Alias Hints
- Floating Point Model
- Security
- Optimizations for MSIL Code
- 64bit Code Generation

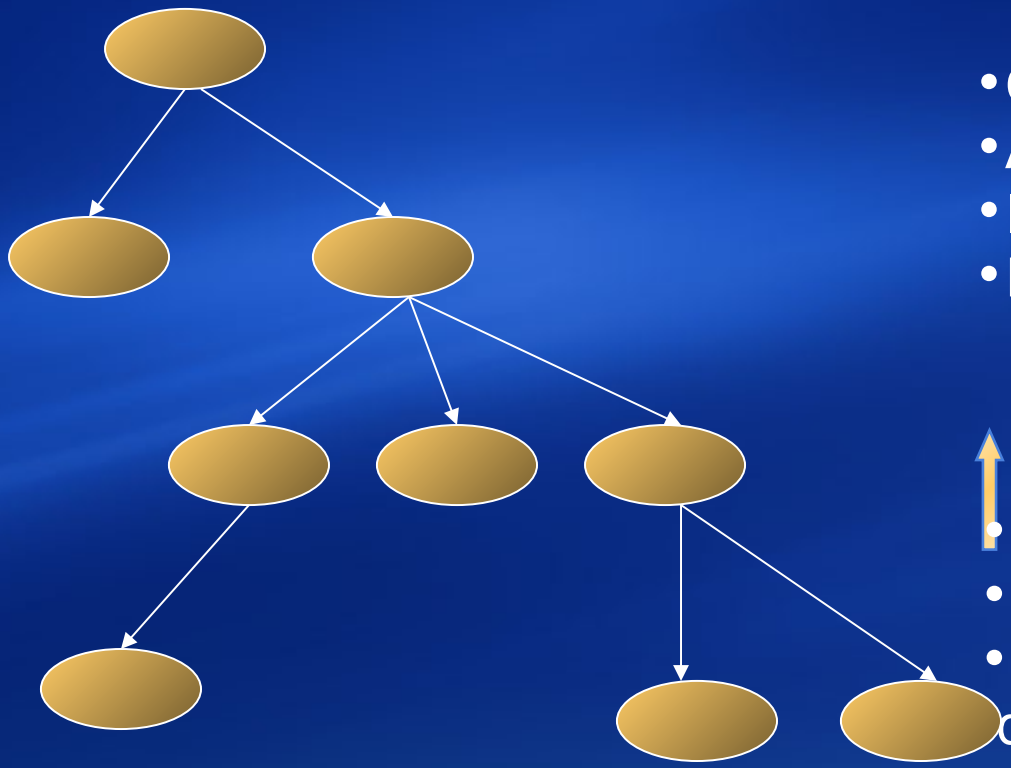
A Refresher Slide And Quick Tips

- There are a some things that many people still don’t do
  - It’s OK to use /O2!
  - /Gy is generally useful (con: slows link time)
    - /DEBUG /INCREMENTAL:NO /OPT:REF /OPT:ICF
- VC++ 2002 and 2003 added lots of code generation improvement
  - Improved support for Pentium4 (/G7)
    - The default is /GB = /G6 in VC2003
  - Addition of /arch:SSE[2] switch

Refresher (cont): WPO

- Whole Program Optimization was introduced in VC++ 2002
- Basic Idea: Do code generation on the whole program at one time to allow analysis to be done on whole program
- Use /GL when compiling and /LTCG when linking
- Enables better inlining opportunities, stack alignment, and custom calling conventions
- We've done work in Whidbey to improve this…

Improved Whole Program Optimization



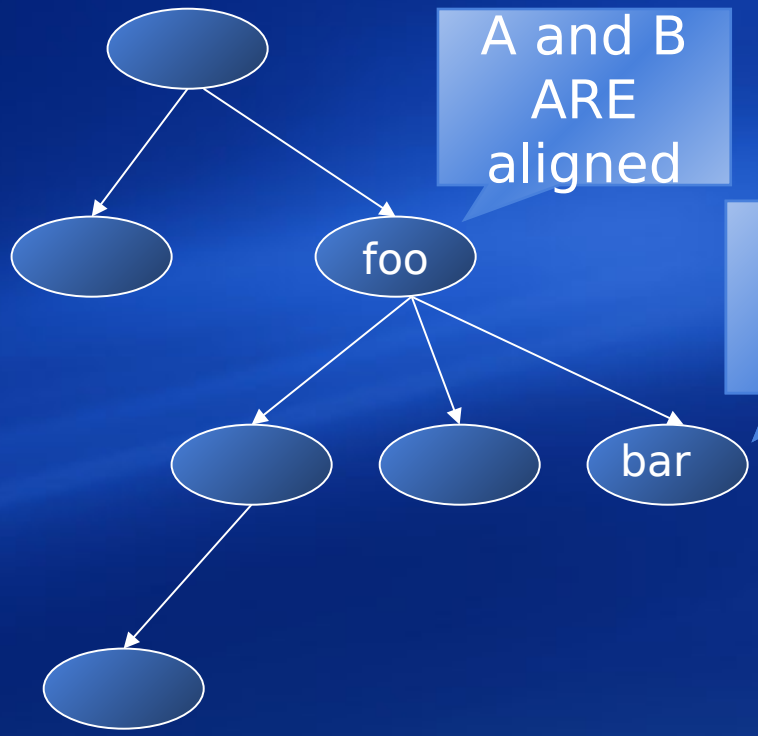
Top Down

- Proving “Theorems”
- Which formals are declspec restrict) pointers
- Constant parameters
- Alignment of data params
- Invariant globals
- Propagate ranges down

Bottom Up

- Creating “Theorems”
- Which globals are changed
- Check if pointer is only dereferenced
- Generate conditionals for range propagation
- Find global restrict pointer

Improved Whole Program Optimization (Example)




```
foo() {
   double A[SIZE1];
   double B[SIZE2];
   ...
   bar(A, B);
   ...
}
void bar(double *A, double *B) {
   ...
   memcpy(A, B, count);
   ...
}
```

A simple example, but indicative of what we do

Profile Guided Optimization?

- Static analysis of code leaves many open questions for the compiler...

```
if(a < b)
   foo();
else
   baz();
```

How often is a < b?

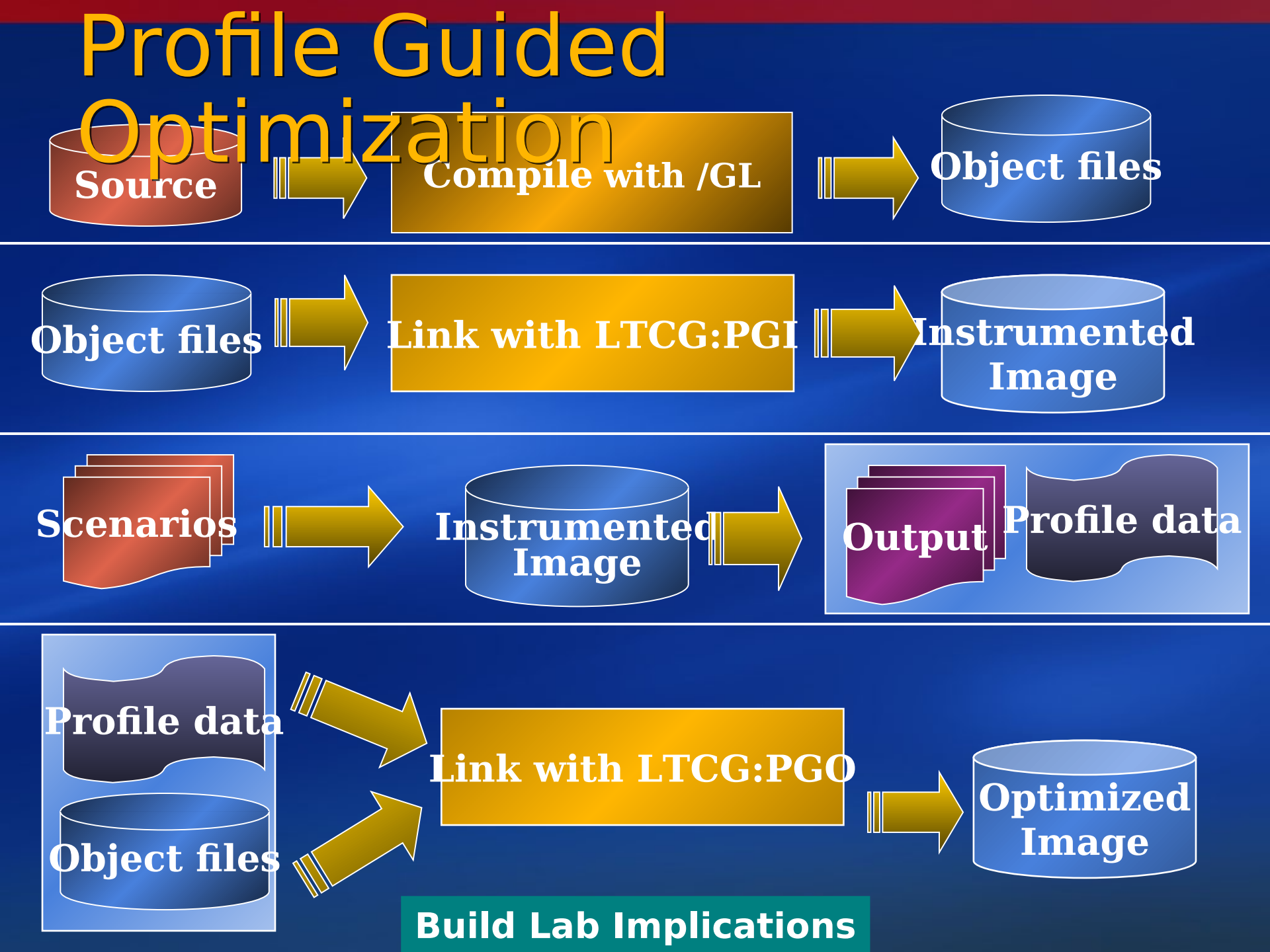
```
for(i = 0; i < count; ++i)
   bar();
```

What is the typical value of count?

Instrumentation

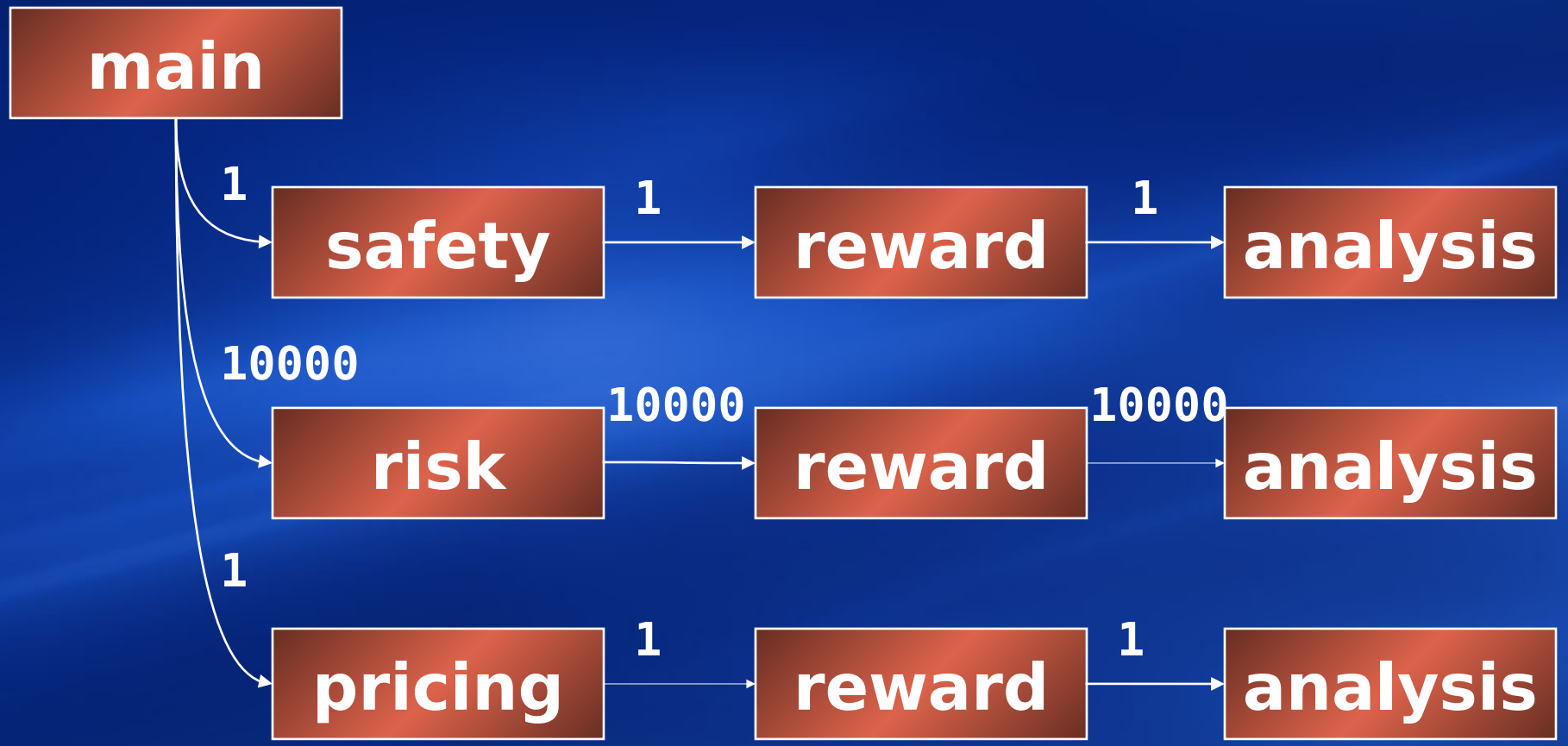
- We instrument with “probes” inserted into the code
- Two main types of probes
  - Value probes
    - Used to construct histogram of values
  - Count (simple/entry) probes
    - Used to count number of times a path is taken
- We try to insert the minimum number of probes to get full coverage
  - Minimizes the cost of instrumentation

Profile Guided Optimization



Build Lab Implications

PGO Demo Call Tree



PGO Whidbey Demo

demo

PDC03

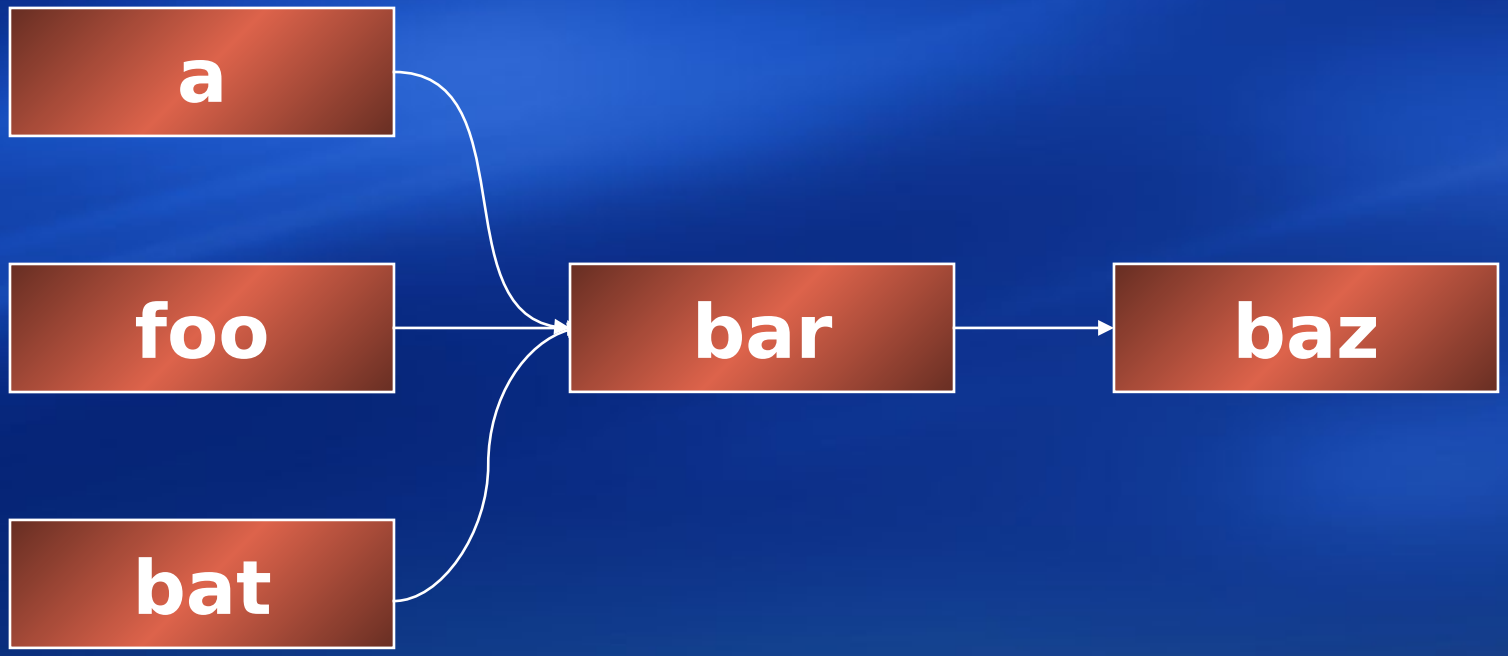
Make the connection

Some of the Optimizations of PGO

- Switch expansion
- Better inlining decisions
- Function/basic block layout
- Cold code separation
- Virtual call speculation
- Partial inlining

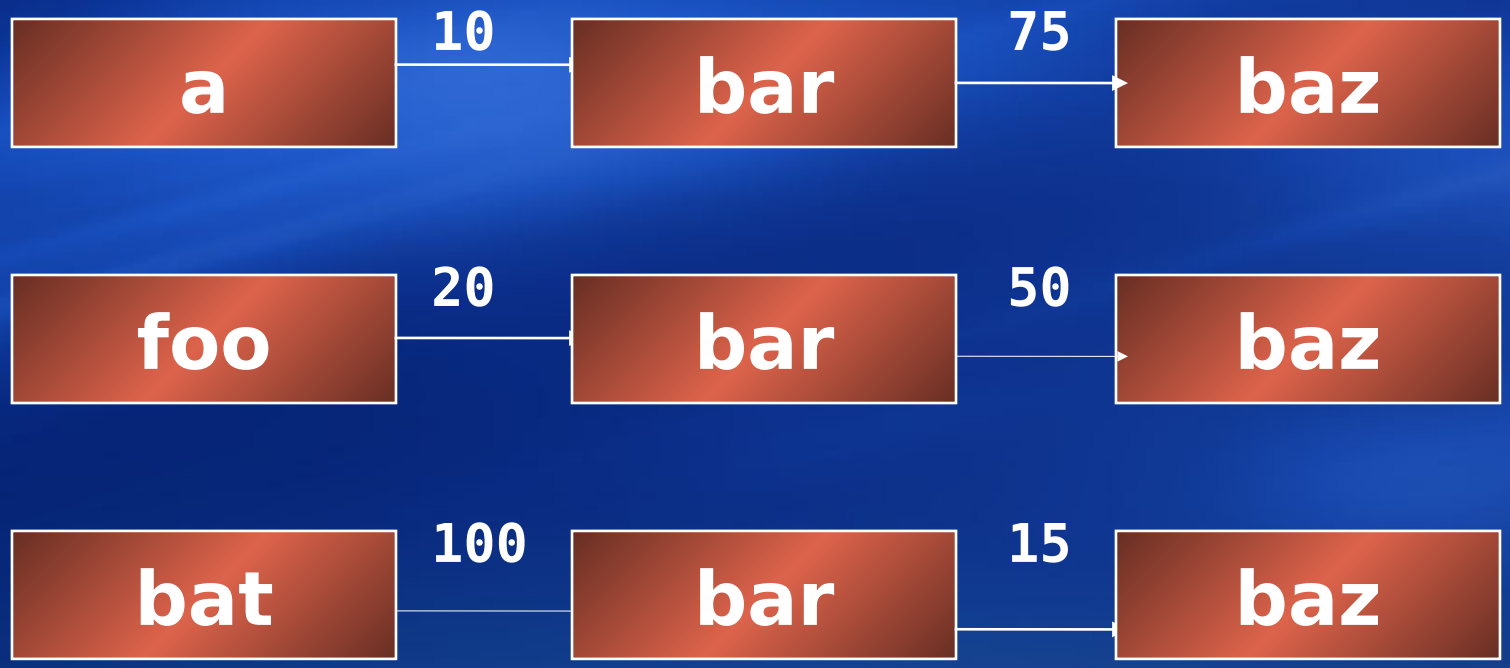
Inlining

Profile Guided uses call graph path profiling.



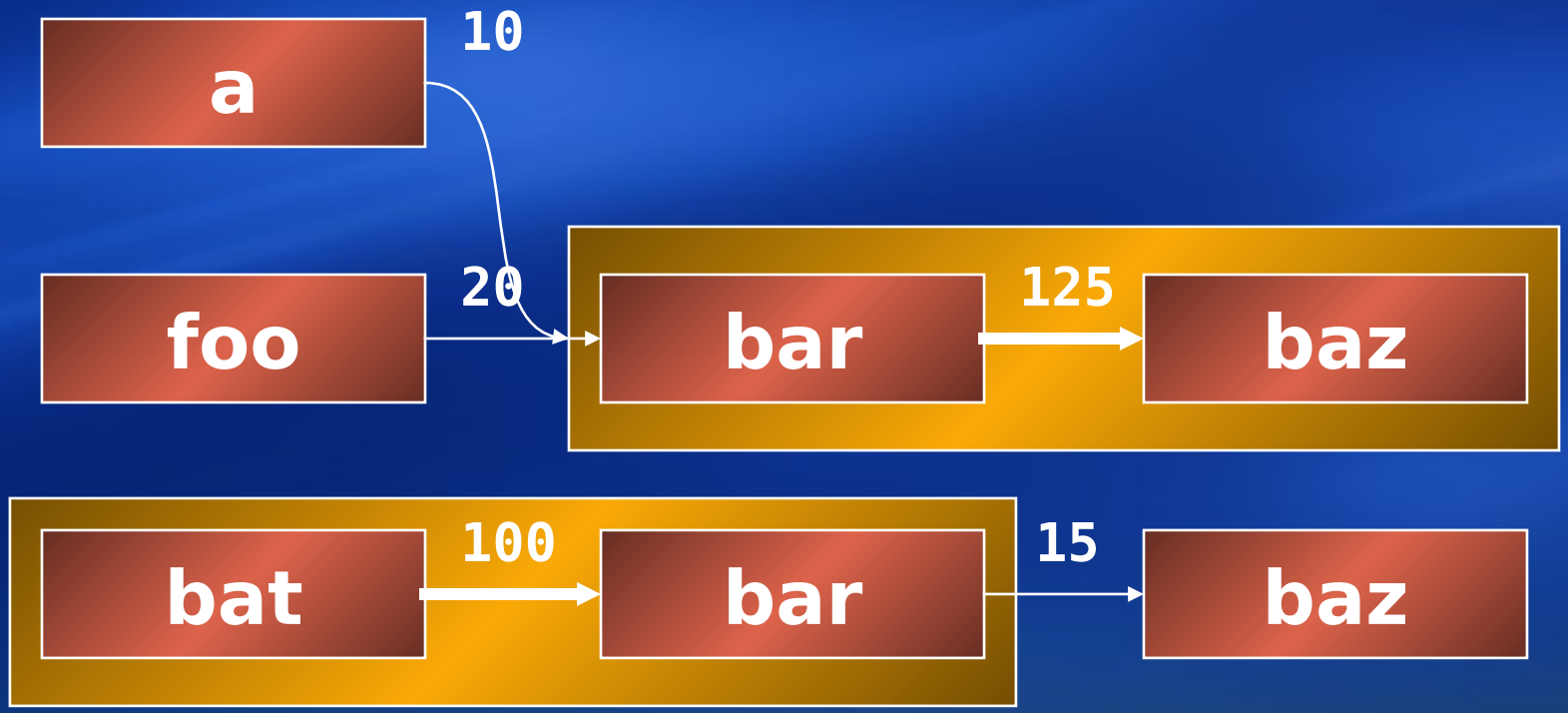
Inlining

Profile Guided uses call graph path profiling.



Inlining (cont)

Inlining decisions are made at each call site.



Wins: Switch Expansion

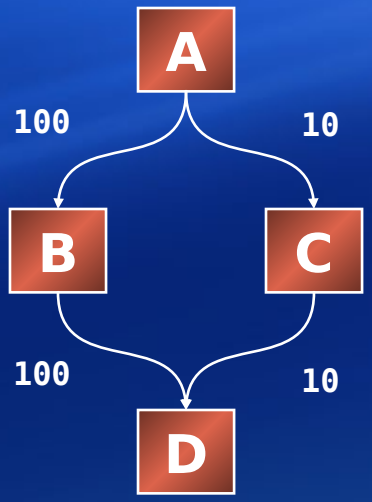
Most frequent values are pulled out.

```
// 90% of the
// time i = 10;
switch (i) {
        case 1: …
        case 2: …
        case 3: …
        default:…
}
```

```
if (i == 10)
        goto default;
switch (i) {
        case 1: …
        case 2: …
        case 3: …
        default:…
}
```

Run building

Basic blocks are ordered so that most frequent path falls through.

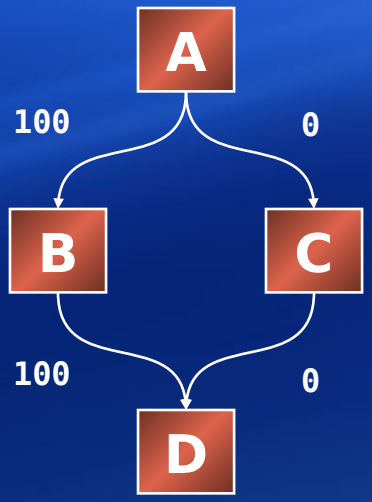


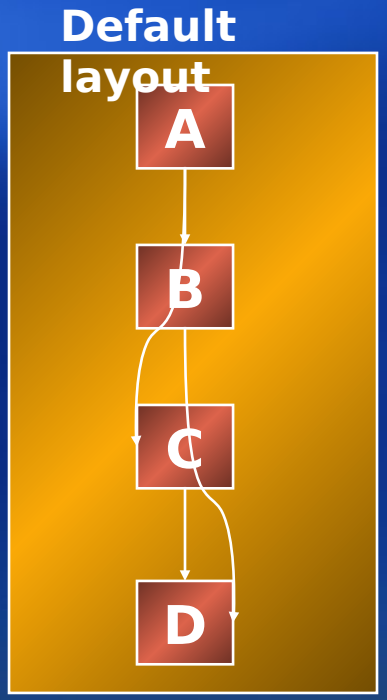
Default layout

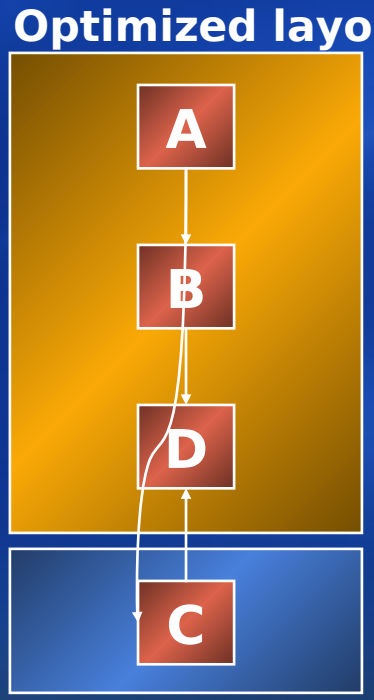
Optimized layout

Separation

Cold functions/blocks are placed in a special section.

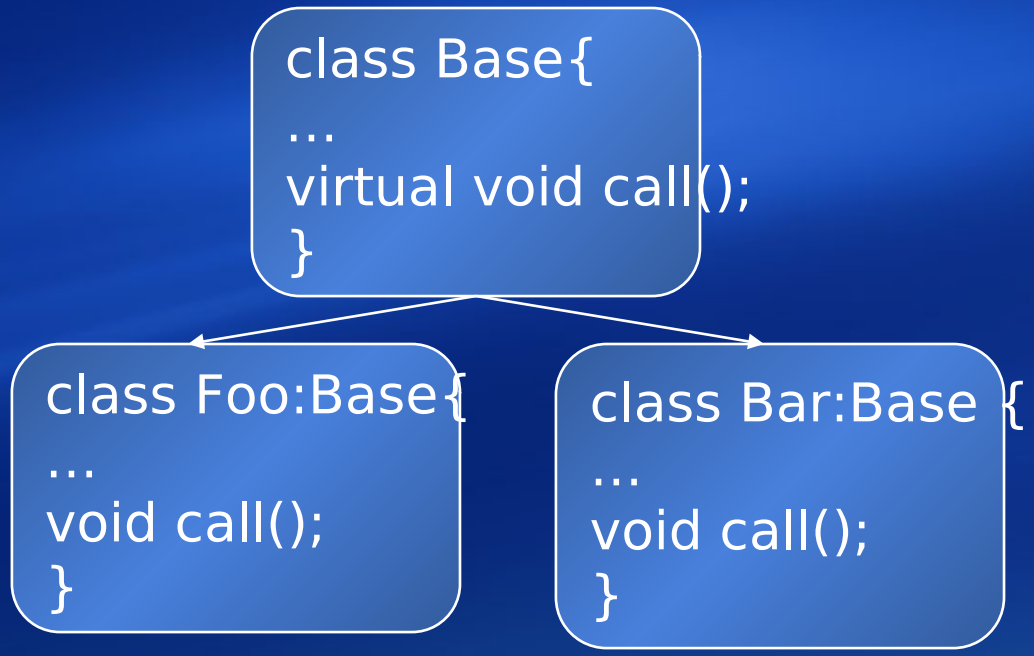






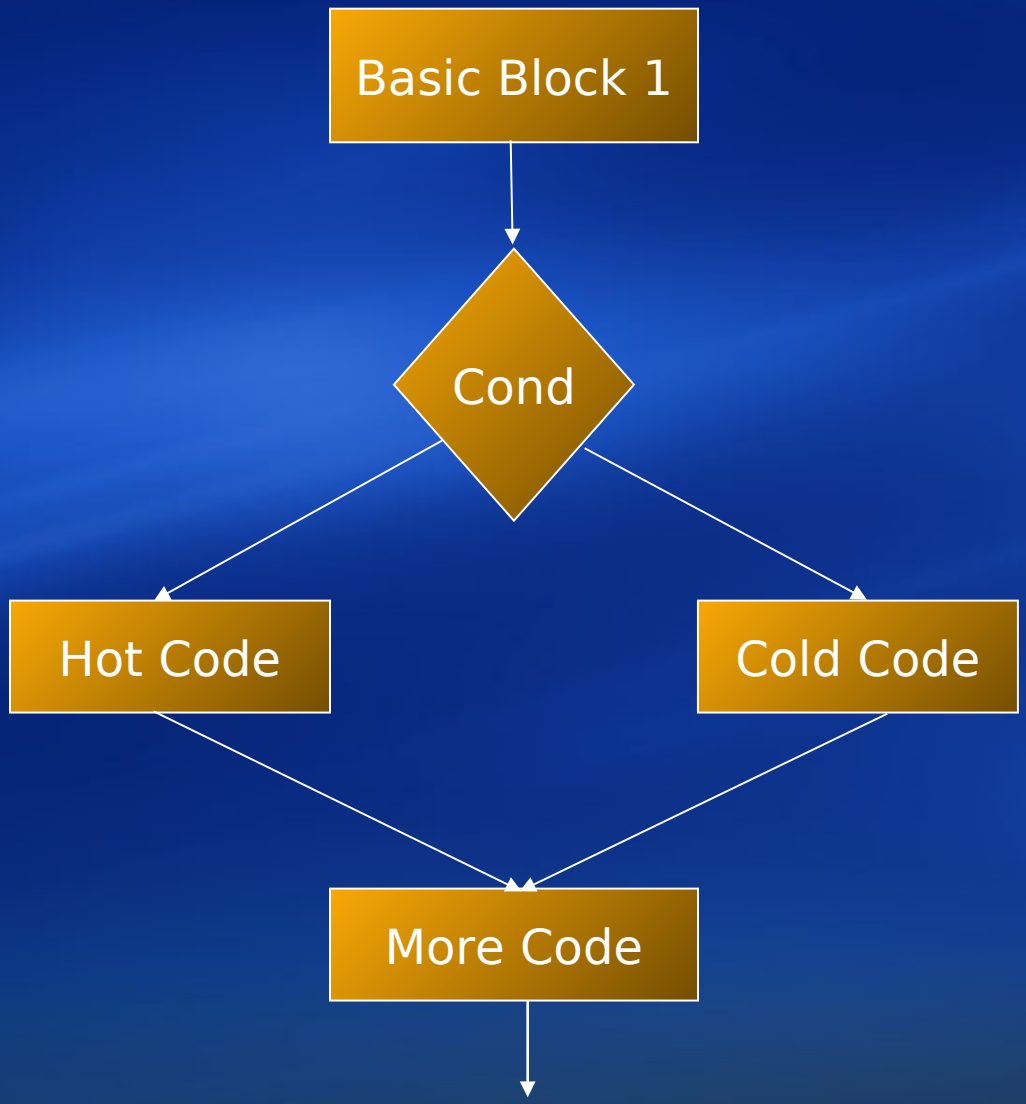
Virtual Call Speculation

The type of object A in function Bar was almost always Foo via the profiles

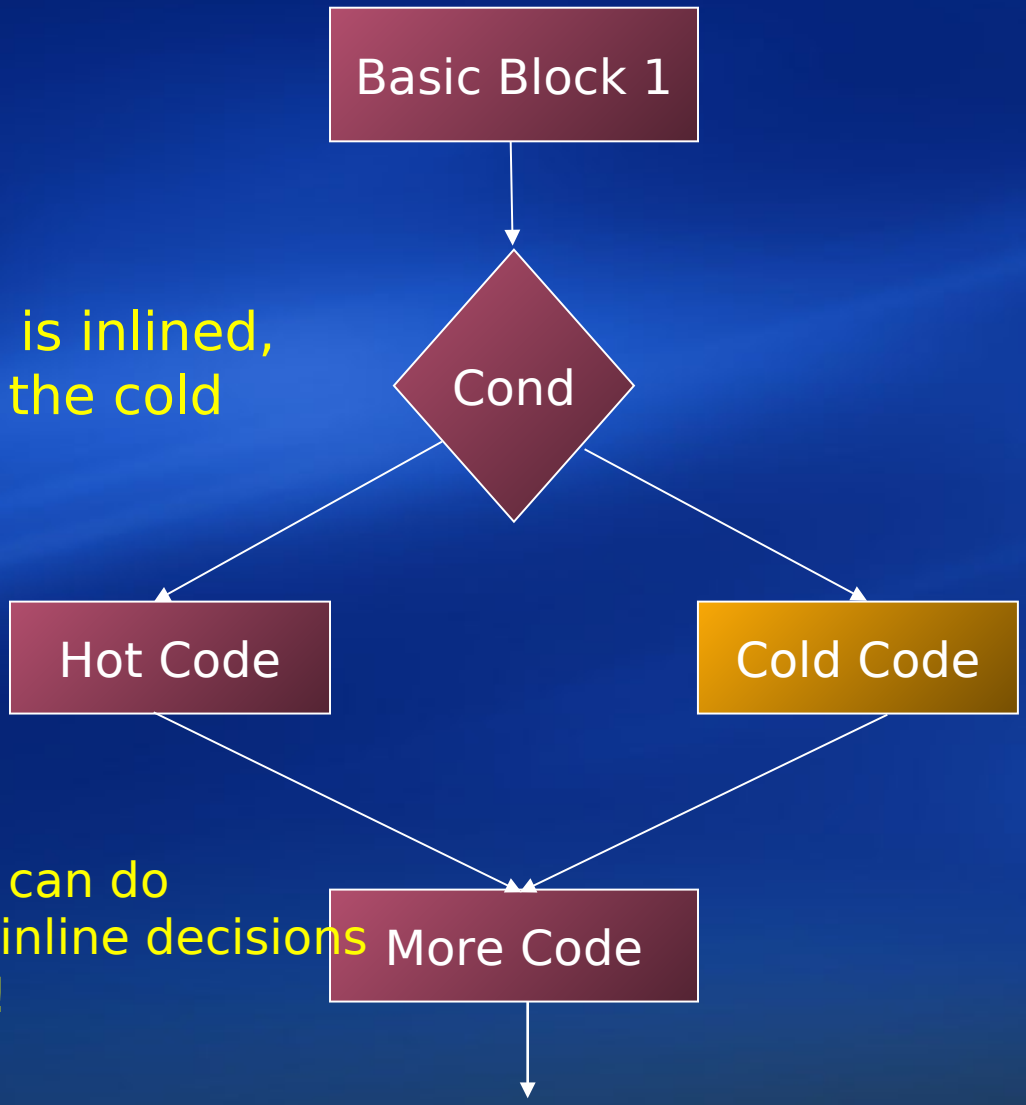



```
void Bar(Base *A)
{
  ...
  while(true)
  {
    ...
    if(type(A) == Foo:Base)
    {
      // inline of A->call();
    }
    else
      A->call();
    ...
  }
}
```

Partial Inlining



Partial Inlining



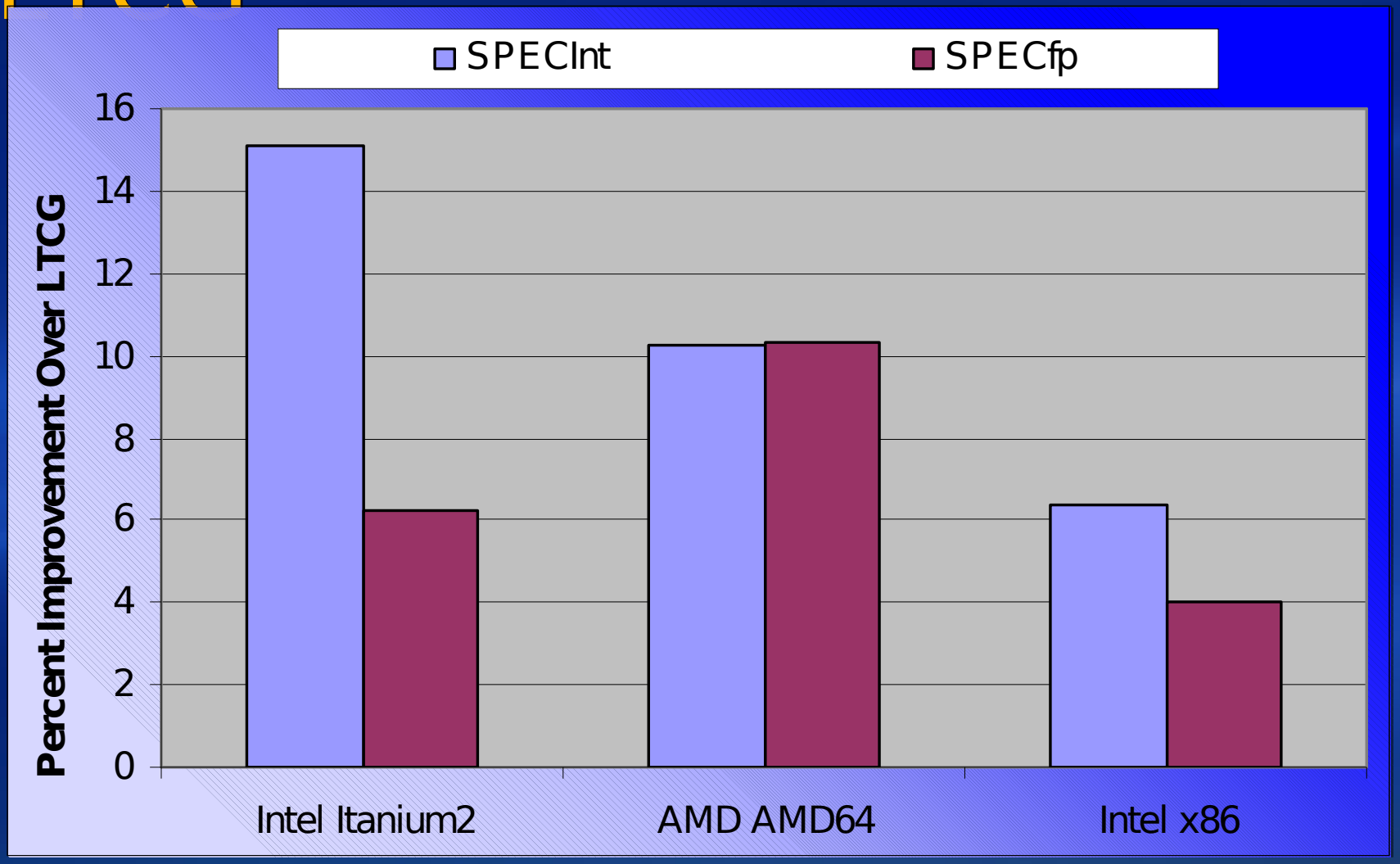
Hot path is inlined,
but NOT the cold

Additionally, we can do
different partial inline decisions
at each call site!

How Much Performance Does PGO Buy You?

- Performance increase is architecture and application specific
  - Itanium benefits the most
  - Large applications tend to benefit more than small
- If you understand your real-world scenarios then PGO is almost always a win

PGO Improvement Over LTCG



OpenMP

- A specification for multithreaded programs
  - Helps hyperthreading as well
- It consists of a set of simple #pragmas and runtime routines
  - #pragma omp parallel
- A common technique:
  - Start with sequential code and parallelize by adding #pragmas
- Most value, where?
  - Parallelizing large loops without loop-dependencies

OpenMP Parallelization

```
void test(int first, int last) {
    #pragma omp parallel for
    for (int i = first; i <= last; ++i) {
        a[i] = b[i] + c[i];
    }
}
```

That's it… the code is parallelized!

```
void test(int first, int last) {
    _vcomp_fork(1, 2, $test$OMP$1, first,
last);
}
```

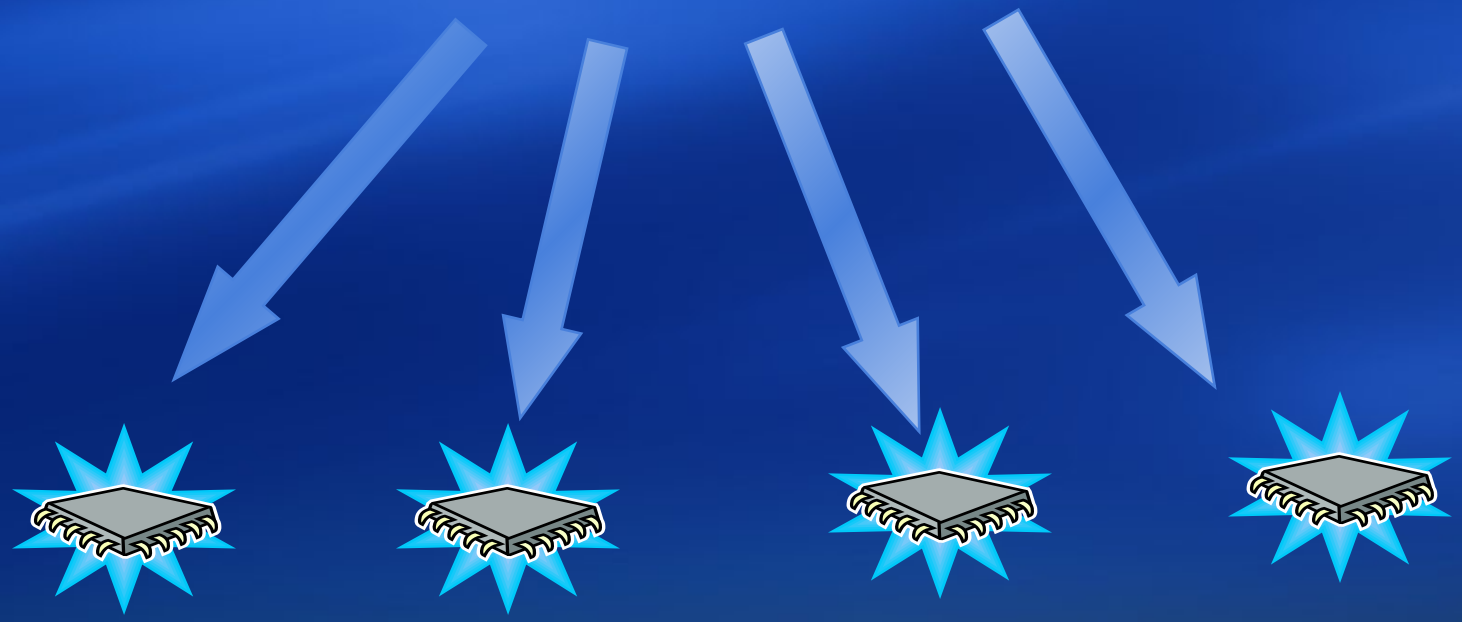
```
void $test$OMP$1(int &first, int
&last) {
    int lower, upper;
    if (first <= last) {

_vcomp_for_static_simple_init(first,
          last, 1, 1, &lower, &upper);
        for (int i = lower; i <= upper;
++i)
        {
            a[i] = b[i] + c[i];
        }
        _vcomp_for_static_end();
        _vcomp_barrier();
```

OpenMP Parallelization

```
void test(int first, int last) {
    #pragma omp parallel for
    for (int i = first; i <= last; ++i) {
        a[i] = b[i] + c[i];
    }
}
```

first = 1
last = 1000



1 ≤ i ≤ 250 | 251 ≤ i ≤ 500 | 501 ≤ i ≤ 750 | 751 ≤ i ≤ 1000

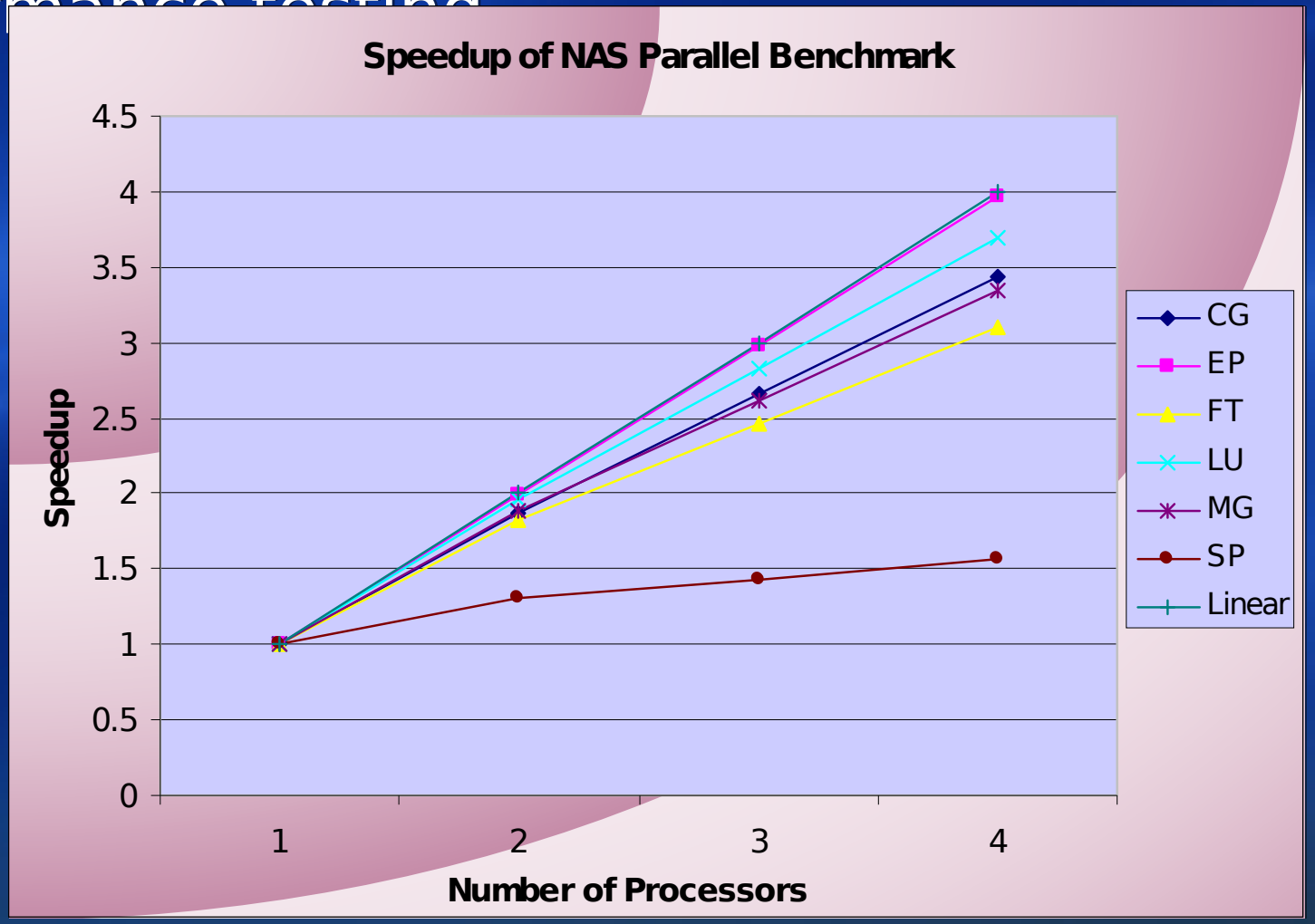
OpenMP for Straight Line Code

```
int bar(int x, int y) {
int a, b, c;
   if(x < 0)
     a = foo(x);
   else
      a = x + 5;
   b = bat(y);
   c = baz(x + y);
   return a*b+c;
}
```

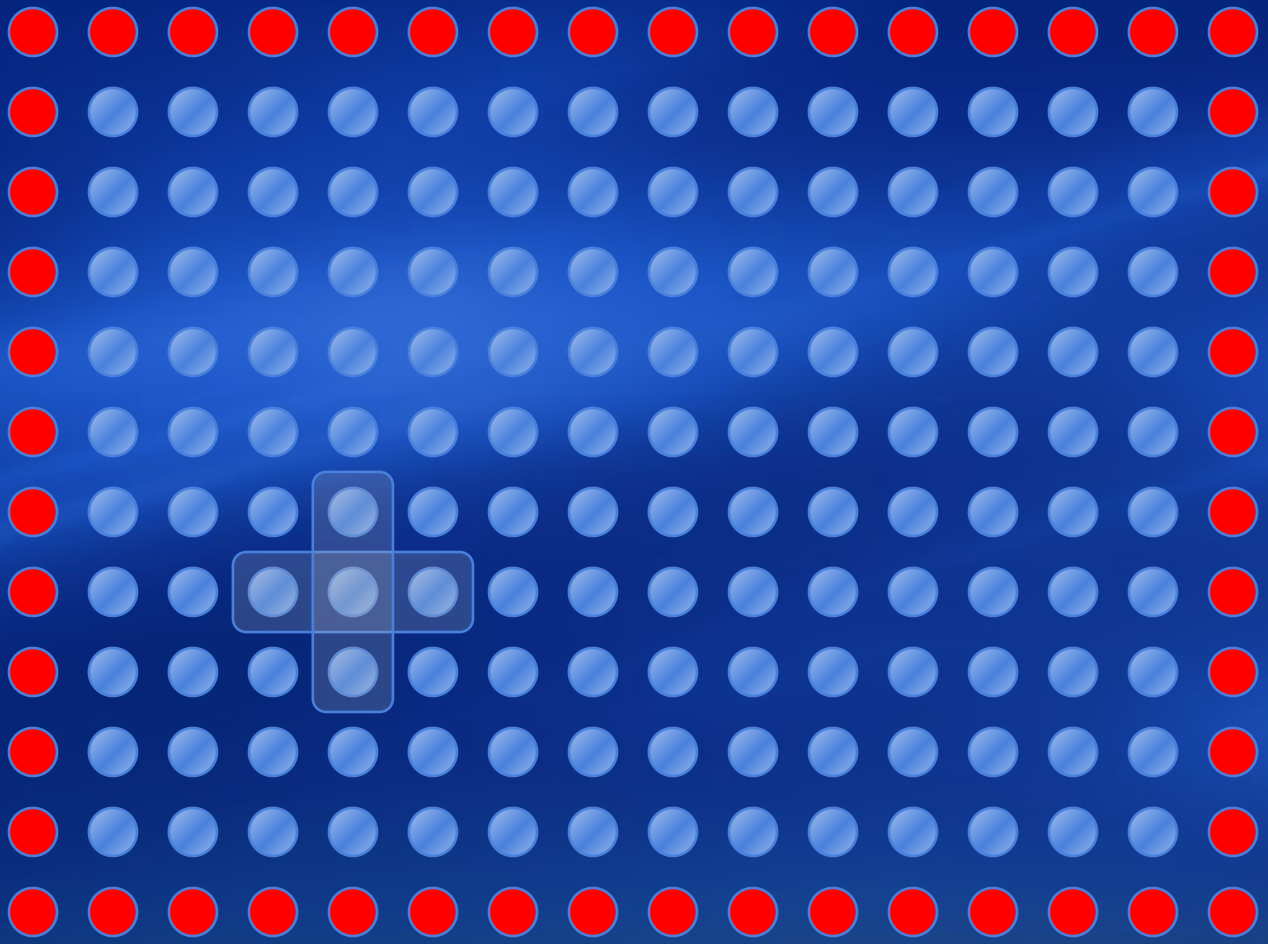
```
int bar(int x, int y) {
int a, b, c;
   #pragma omp parallel sections
   {
      #pragma omp section
      if(x < 0)
        a = foo(x);
      else
         a = x + 5;
      #pragma omp section
      b = bat(y);
      #pragma omp section
      c = baz(x + y);
   }
   return a*b+c;
}
```

NAS Parallel Benchmark with OpenMP

- HPC industry standard benchmark
- We've only begun tuning and performance testing



OpenMP Demo



OpenMP Demo

demo

PDC03

Make the connection

Generating Efficient Code

- Programmer knows a and b don't overlap

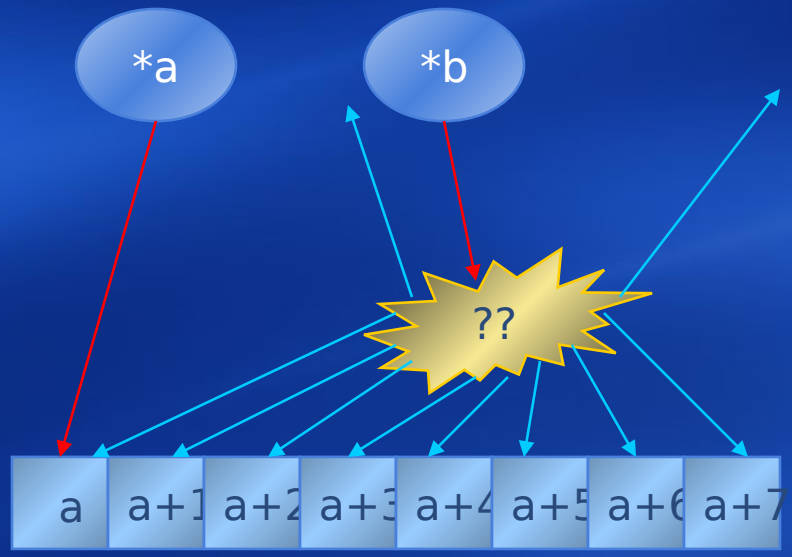
```
void copy8(int * a, int * b) {
        a[0] = b[0];
        a[1] = b[1];
        a[2] = b[0];
        a[3] = b[1];
        a[4] = b[0];
        a[5] = b[1];
        a[6] = b[0];
        a[7] = b[1];
}
```

```
ecx = a, eax = b
mov     edx, DWORD PTR [eax]
mov     DWORD PTR [ecx], edx
mov     edx, DWORD PTR [eax+4]
mov     DWORD PTR [ecx+4], edx
mov     edx, DWORD PTR [eax]
mov     DWORD PTR [ecx+8], edx
mov     edx, DWORD PTR [eax+4]
mov     DWORD PTR [ecx+12], edx
mov     edx, DWORD PTR [eax]
mov     DWORD PTR [ecx+16], edx
mov     edx, DWORD PTR [eax+4]
mov     DWORD PTR [ecx+20], edx
mov     edx, DWORD PTR [eax]
mov     DWORD PTR [ecx+24], edx
mov     eax, DWORD PTR [eax+4]
mov     DWORD PTR [ecx+28], eax
```

Generating Efficient Code

- Programmer knows a and b don't overlap

```
void copy8(int * a, int * b) {
        a[0] = b[0];
        a[1] = b[1];
        a[2] = b[0];
        a[3] = b[1];
        a[4] = b[0];
        a[5] = b[1];
        a[6] = b[0];
        a[7] = b[1];
}
```



Aliasing

- The problem is known as “aliasing”
- Don’t know what a pointer can point to
- /Oa is removed
  - A death sentence for those that tried to use it
- New keyword added “__restrict”
  - Promise made to compiler: address pointed to by restrict pointer can NOT be accessed through other pointers
  - Similar to the C99 ‘restrict’ keyword
  - Example Usage: int * __restrict

__restrict – A compiler hint

- Programmer knows a and b don't overlap

```
void copy8(int * __restrict a, int * b) {
        a[0] = b[0];
        a[1] = b[1];
        a[2] = b[0];
        a[3] = b[1];
        a[4] = b[0];
        a[5] = b[1];
        a[6] = b[0];
        a[7] = b[1];
}
```

```
        eax = a, edx = b
mov     ecx, DWORD PTR [edx]
mov     edx, DWORD PTR [edx+4]
mov     DWORD PTR [eax], ecx
mov     DWORD PTR [eax+4], edx
mov     DWORD PTR [eax+8], ecx
mov     DWORD PTR [eax+12], edx
mov     DWORD PTR [eax+16], ecx
mov     DWORD PTR [eax+20], edx
mov     DWORD PTR [eax+24], ecx
mov     DWORD PTR [eax+28], ed
```

Floating Point Model

- /Op made your code run slow
  - No intermediate switch
- New Floating Point Model
  - /fp:fast
  - /fp:precise (default)
  - /fp:strict
  - /fp:except

/fp:fast

- Used when performance matters most
- Used when you know your application is not in the dark corners of FP world
- What can /fp:fast do?
  - Association
  - Distribution
  - Factoring inverse
  - Scalar reduction
  - Copy propagation

/fp:precise

- The default floating point switch
- The sweet spot…
  - Performance and Precision
    - A few percent slower than /fp:fast
- IEEE Conformant (sans exceptions)
  - Hoisting of expressions (even if they can raise exceptions)
- At assignments, casts, function calls this will round to the appropriate precision
  - Sometimes requires the writing of a value from register to memory (homing)

/fp:except and /fp:strict

- /fp:except[-]
  - Reliable floating point exceptions
    - Thrown and not thrown when expected
      - Faults and traps, when reliable, should occur at the line that causes the exception
      - FWAITs on x86 might be added
    - AMD64 and IA/64 are reliable at the hardware level
    - Not compatible with /fp:fast
    - Not compatible with /clr
- /fp:strict
  - The strictest FP option
    - Turns off contractions
    - Assumes floating point control word can change or that the user will examine flags
  - /fp:except is implied
  - Low double digit percent slowdown versus /fp:fast
  - Not compatible with /clr

Floating Point Precision Characteristics

	X86	AMD64	Itanium
Default Precision (significand)	53bit	Based on source/dest precision (uses SSE[2])	64bit
/fp:fast	Will not home, may spill	Will choose fastest	Will not use completer
/fp:precise	Will home, except when assigning to double	Has appropriate sized operations	Will use completer for precision at rounding point
/fp:strict	Always home	Has appropriate sized operations	Will use completer for precision on each operation

Floating Point Example

```
double a, b, c, d;
float x, m, z;

d = (a + b) * (c + x);
z = (x + m) * m;
```

a + b evaluated with “default” precision

x is promoted up and c + x is evaluated with “default” precision

/fp:precise,fast – the final result is left in a register.

/fp:strict – the final result is rounded to d

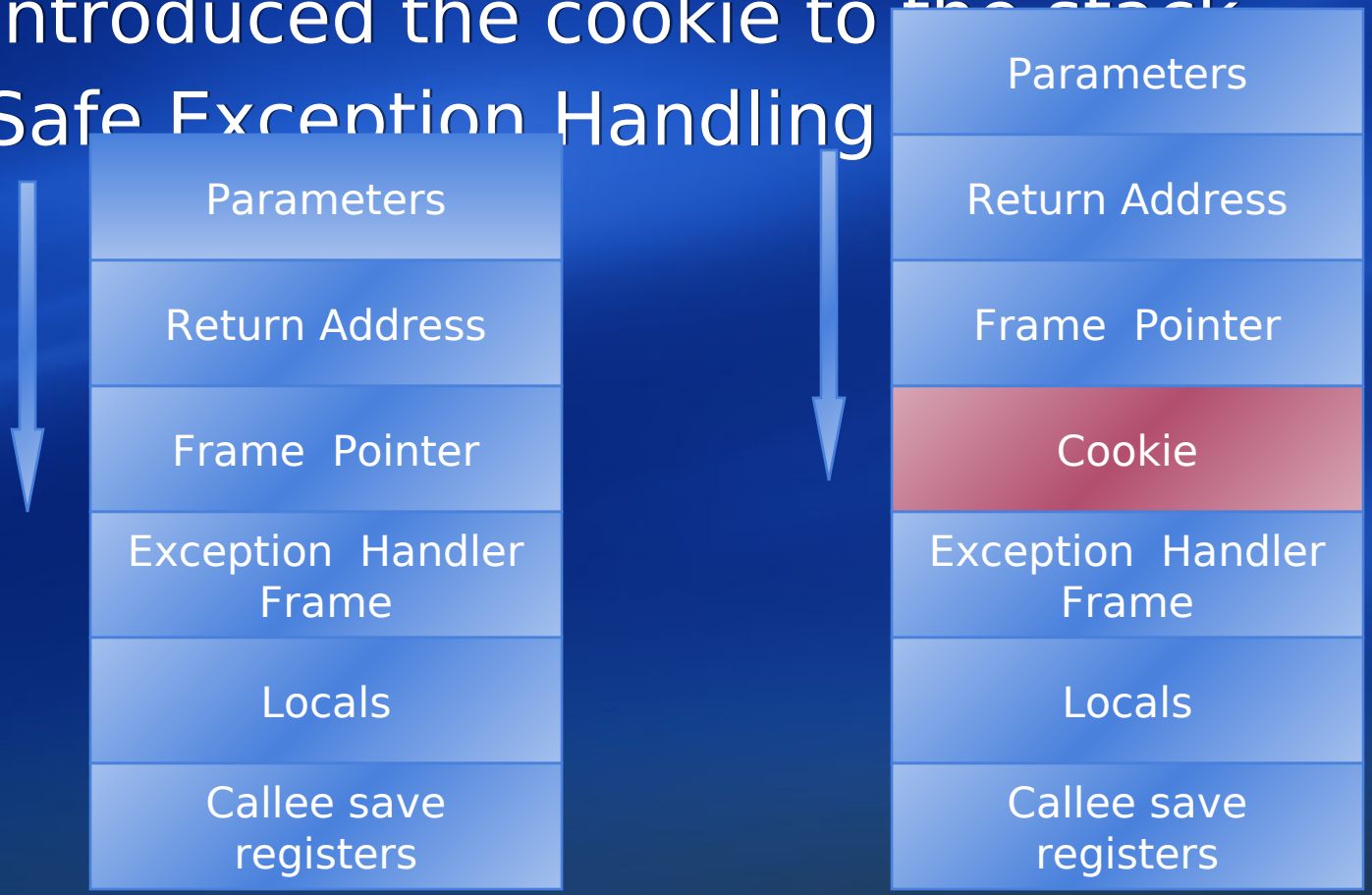
/fp:precise,fast – x + m evaluated with “d

/fp:fast – the final result, z, is left in a register

/fp:precise,strict – the final result is rounded to a float

Better Security for Generated Code

- /GS introduced in 7.0 and improved in 7.1
- Helped prevent attacking the return address
- Introduced the cookie to the stack
- Safe Exception Handling



Better Security for Generated Code

- We continue this improvement in security for Whidbey

- Pointer parameters are vulnerable

Better Security for Generated Code

- Vulnerable parameters are now safe

C++ Optimizer Meets MSIL

- Optimization at the MSIL level (prior to JITing)
  - Does your standard optimizations including expression optimization and global optimization
  - Where does it have problems?
    - Strength reduction
    - Inlining
  - New Types of Decisions
    - Loop Unrolling
    - Inlining
- “Leveraging Native Code with .NET” – TLS 311… Wednesday@11:30am

64bit Code Generation

- x86 is no longer the only desktop/server platform
- Visual C++ Compiler for Itanium and AMD64
  - Cross Compilers and 64bit Hosted Compilers
- Get your code 64bit clean
  - int and long are 32bit, but pointers are 64bit
  - Common pitfalls:
    - Truncation
    - Alignment

64bit Whidbey Demo

demo

PDC03

Make the connection

Take Aways

- Generating Quality Code is What Visual C++ Does
  - Understand scenarios necessary for PGO
  - OpenMP, the easiest way to write multithreaded code
  - Alias hints with __restrict can improve code
  - New FP model has performance and accuracy
  - /GS will help prevent even more attacks
  - MSIL code with VC++ will be faster
  - Get your code 64bit clean now

For More Information

- Hands-on-lab here at the PDC
- Ask the Experts (Hall F/G)
  - Tuesday 6:30pm-9:00pm
- VC++ Demo Booth (Product Pavilion)
  - Several VC people there and walking around conference… talking to you is their top priority
- 64 bit Code Cleaning:
  - http://msdn.microsoft.com/msdnmag/issues/01/11/XP64/default.aspx
- OpenMP specification located at:
  - http://www.openmp.org
- PDC Newsgroups
  - http://msdn.microsoft.com/vstudio/whidbeypdc
- My email: kanggatl@microsoft.com

Microsoft®